deep se
dependable evolvable pervasive software engineering group

Constructors

Federica Filippini - Danilo Ardagna

Politecnico di Milano
federica.filippini@polimi.it
danilo.ardagna@polimi.it

POLITECNICO DI MILANO

Content

- Constructors
  - Default constructors
  - Initializer List
  - Copy and Assignment
- Defining a type member

Constructors ~ special methods of a class

- Each class has to define how objects of its type can be initialized
- Classes control object initialization by defining one or more special member functions known as **constructors**
  - The job of a constructor is to initialize the data members of a class object
  - A constructor is run whenever an object of a class type is created

Constructors

- Constructors have the same name as the class
- Unlike other functions, constructors have no return type
- Like other functions, constructors have:
  - a (possibly empty) parameter list
  - a (possibly empty) function body

Constructors

- A class can have multiple constructors (overloaded methods)
  - must differ from each other in the number or types of their parameters
- Unlike other member functions, constructors may not be declared as const
  - when we create a const object of a class type, the object does not assume its "constness" until after the constructor completes the object's initialization
  - constructors can write to const objects during their construction

Sales_data Class

```
class Sales_data {
public:
     std::string isbn() const {return bookNo;}
     Sales_data& operator+=(const Sales_data&);
private:
     std::string bookNo;
     unsigned units_sold = 0;
     double revenue = 0.0;
};
```

In-class initializer

apparently we have no constructors (∄ method with the same name of the class)

we're assigning a value to 2 members of the class

Default Constructors

- Classes control default initialization by defining a special constructor, known as the **default constructor:**
  - takes **no arguments**
  - is special in various ways, one of which is that if our class does not explicitly define any constructors, it will be implicitly defined by the compiler
- The compiler-generated constructor is known as the **synthesized default constructor.** For most classes, this synthesized constructor initializes each data member of the class as follows:
  - If there is an in-class initializer, use it to initialize the member
  - Otherwise, default-initialize the member

Because Sales_data provides initializers for units_sold and revenue, the synthesized default constructor uses those values to initialize those members. It default initializes bookNo to the empty string.

We cannot always rely on the Synthesized Default Constructor

- Only fairly simple classes can rely on the synthesized default constructor
- The compiler generates the default for us only if we do not define any other constructors
  - If we define any constructors, the class will not have a default constructor unless we define that constructor ourselves explicitly
  - Rationale: if a class requires control to initialize an object in one case, then the class is likely to require control in all cases

We cannot always rely on the Synthesized Default Constructor

- For some classes, the synthesized default constructor does the wrong thing:
  - E.g., objects of built-in or compound type (such as arrays and pointers) have undefined value when they are default initialized
  - We should **initialize those members inside the class** or **define our own version of the default constructor**
    - Otherwise, we could create objects with members that have undefined value
- Sometimes the compiler is unable to synthesize one
  - E.g., if a class has a member that has a class type, and that class doesn't have a default constructor, then the compiler can't initialize that member

for example an un-initialized pointer is initialized to point something random → we don't want to do that

The Role of the Default Constructor

```
class NoDefault {
public:
     NoDefault(const std::string&);
     // additional members follow, but no other constructors
};

struct A {  // my_mem is public by default;
     NoDefault my_mem;
};

A a;  // error: cannot synthesize a constructor for A
```

constructor

(because one of the member has no default constructor)

- In practice, it is almost always right to provide a default constructor if other constructors are being defined

The Role of the Default Constructor

- The default constructor is used automatically whenever an object is default or value initialized
- Default initialization happens when:
  - we define non-static variables or arrays at block scope without initializers
  - a class that itself has members of class type uses the synthesized default constructor
  - members of class type are not explicitly initialized in a constructor initializer list
- Value initialization happens:
  - during array initialization when we provide fewer initializers than the size of the array
  - when we define a local static object without an initializer
  - when we explicitly request value initialization by writing an expressions of the form T() where T is the name of a type (e.g., vector)

Classes must have a default constructor in order to be used in these contexts

Defining the `Sales_data` Constructors

- We'll define three constructors with the following parameters:

1. A const string& representing an ISBN, an unsigned representing the count of how many books were sold, and a double representing the price at which the books sold
2. A const string& representing an ISBN. This constructor will use default values for the other members
3. An empty parameter list (i.e., the default constructor), which we must define because we have defined other constructors

Defining the `Sales_data` Constructors

```
Class Sales_data {
   public:
3.    Sales_data() = default;
2.    Sales_data(const std::string &s): bookNo(s) { }
1.    Sales_data(const std::string &s, unsigned n, double p):
            bookNo(s), units_sold(n), revenue(p*n) { }

      // other members as before
      std::string isbn() const { return bookNo; }
      Sales_data& operator+=(const Sales_data&);
      double avg_price() const;
   private:
      std::string bookNo;
      unsigned units_sold = 0;
      double revenue = 0.0;
};
```

This is how we define the DEFAULT CONSTRUCTOR (here we define it because otherwise it would be covered by the other 2 constructors)

ALTERNATIVE (always for the default):

```
Sales_data() { };
```


Defining the `Sales_data` Constructors

```
Class Sales_data {
   public:
      Sales_data() = default;
      Sales_data(const std::string &s): bookNo(s) { }
      Sales_data(const std::string &s, unsigned n, double p):
            bookNo(s), units_sold(n), revenue(p*n) { }

      // other members as before
      std::string isbn() const { return bookNo; }
      Sales_data& combine(const Sales_data&);
      double avg_price() const;
   private:
      std::string bookNo;
      unsigned units_sold = 0;
      double revenue = 0.0;
};
```

Constructor Initializer List — always after the ":"

p*n because the revenue is the price (p) times the number of units sold (n)

Defining the `Sales_data` Constructors

```
Class Sales_data {
   public:
      Sales_data() = default;
      Sales_data(const std::string &s): bookNo(s) { }
      Sales_data(const std::string &s, unsigned n, double p):
            bookNo(s), units_sold(n), revenue(p*n) { }

      // other members as before
      std::string isbn() const { return bookNo; }
      Sales_data& combine(const Sales_data&);
      double avg_price() const;
   private:
      std::string bookNo;
      unsigned units_sold = 0;
      double revenue = 0.0;
};
```

```
Sales_data(const std::string &s):
      bookNo(s), units_sold(0), revenue(0){ }
```

units_sold and revenue receives 0 because it's the in-class initialization

Constructor Initializer List

```
// legal but sloppier way to write the Sales_data
// constructor: no constructor initializers
Sales_data::Sales_data(const string &s,
unsigned cnt, double price)
{
      bookNo = s;
      units_sold = cnt;
      revenue = cnt * price;
}
```

Initialization in the body (it's actually an assignment, the other way is a more proper initialization)

- How significant this distinction is depends on the type of the data member

Constructor Initializer List

- When we define variables, we typically initialize them immediately rather than defining them and then assigning to them:

```
string foo = "Hello World!"; // define and initialize
string bar; // default initialized to the empty string
bar = "Hello World!"; // assign a new value to bar
```

* equivalent to the init. List
equivalent to the initialization in the body (**not** convenient for large objects / containers)

- Exactly the same distinction between initialization and assignment applies to the data members of objects
  - if we do not explicitly initialize a member in the constructor initializer list, that member is default initialized before the constructor body starts executing

Sometimes we cannot use the body of the constructors:

Constructor Initializers are sometimes required

- We can often, but not always, ignore the distinction between whether a member is initialized or assigned:
  - Members that are const or references must be initialized
  - Members that are of a class type that does not define a default constructor also must be initialized

```
class ConstRef {
public:
      ConstRef(int ii);
private:
      int i;
      const int ci; int &ri;
};
```


Constructor Initializers are sometimes required

- The members ci and ri must be initialized. Omitting a constructor initializer for these members is an error:

```
// error: ci and ri must be initialized
ConstRef::ConstRef(int ii)
{     // assignments:
      i=ii; //ok
      ci = ii; // error: cannot assign to a const
      ri = i; // error: ri was never initialized
}
```

this is not an initialization, it's like in * and so ci got a workdown (?) initialization and we cannot change it

- The correct way to write this constructor is:

!!!

```
// ok: explicitly initialize reference and const members
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) { }
```

we cannot assign values to references!

Delegating Constructors

- A delegating constructor uses another constructor from its own class to perform its initialization

We can re-use one constructor to define other constructors. Here we replicate the 3 constructors (that we defined before) with the delegation method

```
class Sales_data {
public:
      // non-delegating constructor initializes members from
      // corresponding arguments
1.    Sales_data(const std::string& s, unsigned cnt, double price):
        bookNo(s), units_sold(cnt), revenue(cnt*price) { }

      // remaining constructors all delegate to another
      // constructor
3.    Sales_data(): Sales_data("", 0, 0) {}
2.    Sales_data(const std::string& s): Sales_data(s, 0,0){}

      // other members as before
};
```

Constructors and initialization order

- Initializers lists are run first but members are initialized in order as they appear in the class declaration (**in some situations this might create mess, use the same order!**)
- Then, (non-static) data members are initialized in order of declaration in the class definition according to in-class initializers
- Finally, the body of the constructor is executed

- If a constructor relies on a delegating constructor, the delegated constructor is executed first, then the control returns to the delegating constructor and its body is executed

DEMO

Copy, Assignment, and Destruction

- Classes also control what happens when we copy, assign, or destroy objects of the class type
- Objects are copied in several contexts:
  - when we **initialize a variable**
  - when we **pass or return** an object **by value**
  - when we use the **assignment operator**
- Objects are destroyed:
  - when they **cease to exist**, such as when a local object is destroyed on exit from the block in which it was created
  - **objects stored in a vector** (or an array) are destroyed when that vector (or array) is destroyed
- If we do not define these operations, the compiler will synthesize them for us
  - Ordinarily, the versions that the compiler generates for us **execute by copying, assigning, or destroying each member of the object**

Copy, Assignment, and Destruction

```
Sales_data total; // variable to hold the running sum
Sales_data trans; // variable to hold data for the next transaction

total = trans;
```

```
// default assignment for Sales_data is equivalent to:
total.bookNo = trans.bookNo;
total.units_sold = trans.units_sold;
total.revenue = trans.revenue;
```

(by the way)
this is what happens with vectors

ARRAYS :

```
int a[10];
int b[10];
a=b;
for(unsigned j=0 ; j<10 ; ++j)
    a[j] = b[j];
```

VECTORS :

```
std::vector<int> aa(10);
std::vector<int> bb(10);
aa = bb;
```

! vectors are classes while arrays are pointers

Copy, Assignment, and Destruction

- Some classes cannot rely on the synthesized versions:
  - the synthesized versions are unlikely to work correctly for classes that allocate resources that reside outside the class objects themselves (e.g., use dynamic memory)
  - use vectors or a strings to manage the necessary storage in the while, we will be back to this

Defining a Type Member

Type Aliases

- A **type alias** is a name that is a synonym for another type. We can define a type alias in one of two ways
- Traditionally, we use a **typedef**

```
typedef double wages;  // wages is a synonym for double
typedef wages base, *p; // base is a synonym for double,
                        // p for double*
```

- C++ 11 introduced a second way to define a type alias, via an **alias declaration**

```
using SD = Sales_data; // SD is a synonym for Sales_data
```

writing <name-of-the-alias> = <name-of-the-type>;

from here on in the code writing "double" or writing "wages" is the same

here we're saying that "base" is a synonim of "wages" which is a synonim of "double" ⇒ "base" is a synonim of "double"
MOREOVER: "p" is synonim of "double*"

from this on in the code writing "SD" is like writing "Sales_data"

Type Aliases

- A type alias is a type name and can appear wherever a type name can appear

```
wages hourly, weekly; // same as double hourly, weekly;
SD item;              // same as Sales_data item
```

hourly, weekly are doubles
item is Sale_data object

Defining a Type Member

```
class Screen {
public:
    typedef std::string::size_type pos;
    Screen() = default; // needed because Screen has another constructor

    // cursor initialized to 0 by its in-class initializer
    Screen(pos ht, pos wd, char c): height(ht), width(wd), contents(ht * wd, c) { }

    char get() const // get the character at the cursor
        { return contents[cursor]; }
    char get(pos r, pos c) const;

private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

Members that define types must appear before they are used

References

- Lippman Chapters 6, 7